Generic H∀SKELL, SPECIFICALLY

Dave Clarke Andres Löh
Institute of Information and Computing Sciences
Utrecht University, P.O. Box 80.089, 3508 TB Utrecht
The Netherlands
{dave,andres}@cs.uu.nl

Abstract **Generic H∀SKELL** exploits the promising new incarnation of generic programming due to Hinze. Apart from extending the programming language Haskell, Hinze-style polytypism offers a simple approach to defining generic functions which are applicable to types of all kinds. Here we explore a number of simple but significant extensions to Hinze's ideas which make generic programming both more expressive and easier to use. We illustrate our ideas with examples.

Keywords: generic programming, polytypism, program transformations, large bananas, Generic Haskell

1. Introduction

Generic or polytypic programming languages provide a medium for implementing functions whose behaviour is defined inductively on the structure of types. Many algorithms can be defined generically. Commonly cited examples include mapping, folding, pretty printing, parsing, and data compression. Once defined, a generic function can be re-used for any type, even types not yet imagined. The dream of having a complete programming language with adequate support for generic programming has come closer to fruition over recent years. Artifacts such as PolyP (Jansson and Jeuring, 1997), G'Caml (Furuse, 2001a), FISh (Jay, 1999), the generic extension for Clean (Alimarine and Plasmeijer, 2001), and of course, Generic H∀SKELL (Clarke et al., 2001), which is based on the work of Hinze (Hinze, 2000c), among others, provide evidence of this progress. Each is limited to some degree, but we believe that the proposal which offers the best returns is Generic H∀SKELL.

Central to Hinze's proposal is a constraint on the form of generic function's type, namely that *polytypic values possess polykinded types*, which

ensures that the cases of a generic function are sufficiently polymorphic and of a compatible form to be combined together when specialising a generic function for *any* Haskell 98 type, including mutually recursive datatypes and nested datatypes. Each Haskell type is first converted to a *structure type* which is defined in terms of a fixed collection of constructors, over which generic functions are defined. Instances of generic functions are then specialised inductively over the target's structure type, with type abstraction, type application, and type-level fixed-point being *always* interpreted as their value-level counterparts.

A significant advantage of Hinze's approach, apart from being applicable to all Haskell 98 types, is its simplicity. A programmer need not possess a strong theoretical background as programming generic functions is relatively straightforward.

For this paper we don a pragmatist's hat. As we gained experience with our Generic H∀SKELL compiler, we have come to realise that the basic mechanisms do not support everything we wish to do in practice. Thus we have designed and implemented a number of extensions which allow not only more reuse of generic functions but also a larger class of generic functions to be implemented. These extensions, which constitute the contribution of this paper, are

- *default cases*, with which one generic function can copy the cases of an existing one, updating or adding cases as required;
- *constructor cases* which allow generic function cases not only for named types, but also for specific constructors; and
- *generic abstractions* which allow generic functions to be defined by abstracting a type variable from an expression which may involve generic functions.

We describe these extensions and their implementation, and give examples to indicate their possible use.

Before explaining the extensions, we first give an overview of generic programming in Generic H∀SKELL, including a description of how generic functions are compiled into plain Haskell.

2. Classic Generic H∀SKELL

Generic H∀SKELL is an extension to the Haskell programming language (Peyton Jones et al., 1999) which offers a means for writing generic functions. Generic functions, that are also known as polytypic functions (Jansson and Jeuring, 1997), are functions defined over the structure of types. Here we give a short introduction to the style of generic

programming employed in Generic H∀SKELL, before giving an overview of the implementation.

2.1 Generic functions

The most general way of declaring datatypes uses the **data** keyword as in the following example:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Tree}\ a & = & \mathit{Leaf} \\ & | & \mathit{IntLeaf}\ \mathit{Int} \\ & | & \mathit{Node}\{\,\mathit{ref} :: a, \mathit{left} :: \mathit{Tree}\ a, \mathit{right} :: \mathit{Tree}\ a\,\}\,. \end{array}$$

The left-hand side of a datatype specifies a new type name, along with the type's parameters. Thus this definition defines the parametric type *Tree a* with parameter *a*. The right-hand side of a datatype may have multiple alternatives, separated by a vertical bar. Each consists of a constructor and a number of fields that may be labelled. The fields may have any type.

Haskell datatypes are thus sums of products, where the arity of the sum or product is unbounded. This makes the datatypes difficult to process generically, as witnessed in the approaches of Jay, Bellè and Moggi (Jay et al., 1998). Hinze, however, adopts a simpler model, dealing only with binary sums and products (Hinze, 2000b). Each Haskell datatype can be realised in this form by firstly replacing the alternatives with the binary sum operator `:+:`. In a similar way the list of fields for a constructor is replaced by a nested tuple constructed using the binary product operator `:*:`, with the special type *Unit* being used when there are no fields. The constructors and field names are then replaced by special type constructors which mark the positions where they occurred in the original type. Explicitly, the following set of type constructors is used to construct an isomorphic type that exhibits the top-level structure of a type:[1]

$$\begin{array}{lcl} \textbf{data}\ a\ \texttt{:+:}\ b & = & \mathit{Inl}\ a \mid \mathit{Inr}\ b \\ \textbf{data}\ a\ \texttt{:*:}\ b & = & a\ \texttt{:*:}\ b \\ \textbf{data}\ \mathit{Unit} & = & \mathit{Unit} \\ \textbf{data}\ \mathit{Con}\ a & = & \mathit{Con}\ a \\ \textbf{data}\ \mathit{Lab}\ a & = & \mathit{Lab}\ a \end{array}$$

[1] Haskell 98 does not allow infix type constructors. To simplify the notation, we use them here anyway, and further assume that both `:+:` and `:*:` are right associative. In the actual implementation, the following types are used instead:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Sum}\ a\ b & = & \mathit{Inl}\ a \mid \mathit{Inr}\ b \\ \textbf{data}\ \mathit{Prod}\ a\ b & = & a\ \texttt{:*:}\ b \end{array}$$

This type is called the *structure type* for the Haskell datatype.

For example, the type *Tree* has structure type:

$$\begin{array}{lcl} \textbf{type}\ \mathit{Tree}^{\circ}\ a & = & \mathit{Con}\ \mathit{Unit} \\ & \texttt{:+:} & \mathit{Con}\ \mathit{Int} \\ & \texttt{:+:} & \mathit{Con}\ (\mathit{Lab}\ a\ \texttt{:*:}\ \mathit{Lab}\ (\mathit{Tree}\ a)\ \texttt{:*:}\ \mathit{Lab}\ (\mathit{Tree}\ a))\,. \end{array}$$

Notice that the recursive instances of *Tree a* are left intact. This is because the specialisation of generic functions follows only the top-level structure of a type, relying on recursion when processing further instances of *Tree a*. Base types, such as *Int* in our example, are unaffected by the translation: they appear in the structure type in the same way as in the original type.

We can write generic functions which can be specialised to all types by giving cases for the structure type constructors (and primitive types) only. An example generic definition is

$$\begin{array}{llcl} \mathit{gmap}\langle \mathit{Unit}\rangle & \mathit{Unit} & = & \mathit{Unit} \\ \mathit{gmap}\langle \texttt{:+:}\rangle\ \mathit{gmapA}\ \mathit{gmapB} & (\mathit{Inl}\ a) & = & \mathit{Inl}\ (\mathit{gmapA}\ a) \\ \mathit{gmap}\langle \texttt{:+:}\rangle\ \mathit{gmapA}\ \mathit{gmapB} & (\mathit{Inr}\ b) & = & \mathit{Inr}\ (\mathit{gmapB}\ b) \\ \mathit{gmap}\langle \texttt{:*:}\rangle\ \mathit{gmapA}\ \mathit{gmapB} & (a\ \texttt{:*:}\ b) & = & (\mathit{gmapA}\ a)\ \texttt{:*:}\ (\mathit{gmapB}\ b) \\ \mathit{gmap}\langle \mathit{Con}\ c\rangle\ \mathit{gmapA} & (\mathit{Con}\ a) & = & \mathit{Con}\ (\mathit{gmapA}\ a) \\ \mathit{gmap}\langle \mathit{Lab}\ l\rangle\ \mathit{gmapA} & (\mathit{Lab}\ a) & = & \mathit{Lab}\ (\mathit{gmapA}\ a)\,. \end{array}$$

This reimplements the function *map*, which is predefined for lists in the Haskell prelude, generically for all datatypes. The cases for *Con* and *Lab* take an extra argument (c and l in the example) that is bound to a descriptor of the constructor or label in question. These descriptors can be used to query information about constructors and labels, such as their names.

If a generic function is to be applicable to types containing primitive types such as *Int* or *Char* or abstract types such as *IO*, then cases for these types must be added to the function definition. The case for *Int*, which is needed for *gmap* to work on our example *Tree* datatype, is given by

$$\mathit{gmap}\langle \mathit{Int}\rangle\ i = i\,.$$

Whenever a generic function is used on a type, that type is implicitly viewed as the corresponding structure type. Thus, in the call

$$\mathit{gmap}\langle \mathit{Tree}\rangle\ (+1) :: (a \to b) \to \mathit{Tree}\ a \to \mathit{Tree}\ b\ ,$$

which could be used to increase all values in an integer tree by one, the instance $\mathit{gmap}\langle \mathit{Tree}\rangle$ is really defined over Tree°. The precise details

of the underlying specialisation process are deferred until Section 2.3.3. Note that above call will only increase integers in the *ref* positions of the nodes of a tree. The integers appearing in the leaves will be treated according to the case for *Int*, thus left unaltered.

Different cases in the definition of *gmap* have different types. All but the case for *Unit* require additional arguments, equal in number to the number of arguments of the type constructor. This is due to the fact that Generic H∀SKELL implements so-called MPC-style generic functions (Hinze, 2000c) which allows generic functions to be used with types of arbitrary kind. A consequence is that generic definitions possess kind-indexed types. So the complete type information for *gmap* is given by the following recurrence:

$$\begin{array}{lcl}
\textbf{type}\ \mathit{GMap}\langle\!\langle\star\rangle\!\rangle\ s\ t & = & s \to t \\
\textbf{type}\ \mathit{GMap}\langle\!\langle\kappa \to \nu\rangle\!\rangle\ s\ t & = & \forall a\ b\,.\,\mathit{GMap}\langle\!\langle\kappa\rangle\!\rangle\ a\ b \\
 & & \qquad \to \mathit{GMap}\langle\!\langle\nu\rangle\!\rangle\ (s\ a)\ (t\ b) \\
\mathit{gmap}\langle t :: \kappa\rangle & :: & \mathit{GMap}\langle\!\langle\kappa\rangle\!\rangle\ t\ t\,.
\end{array}$$

Here a kind-indexed type *GMap* is defined inductively over the structure of kinds. The first line is for type arguments of kind $\star$. The second line extends the type to arguments of higher kind. Its form is a consequence of the theory explained in Hinze's paper. The kind-indexed type *GMap* is then assigned to the generic function *gmap*.

The type required for each of the cases in the definition of *gmap* can be determined by applying $\mathit{GMap}\langle\!\langle\kappa\rangle\!\rangle\ t\ t$ to the type and kind of the particular case. Observing that $\mathit{Unit} :: \star$, $\mathit{Con}, \mathit{Lab} :: \star \to \star$, and $\texttt{:+:}, \texttt{:*:} :: \star \to \star \to \star$, we obtain:

$$\begin{array}{lcl}
\mathit{gmap}\langle \mathit{Unit}\rangle & :: & \mathit{Unit} \to \mathit{Unit} \\
\mathit{gmap}\langle \texttt{:+:}\rangle & :: & (a \to c) \to (b \to d) \to a\ \texttt{:+:}\ b \to c\ \texttt{:+:}\ d \\
\mathit{gmap}\langle \texttt{:*:}\rangle & :: & (a \to c) \to (b \to d) \to a\ \texttt{:*:}\ b \to c\ \texttt{:*:}\ d \\
\mathit{gmap}\langle \mathit{Con}\rangle & :: & (a \to b) \to \mathit{Con}\ a \to \mathit{Con}\ b \\
\mathit{gmap}\langle \mathit{Lab}\rangle & :: & (a \to b) \to \mathit{Lab}\ a \to \mathit{Lab}\ b\,.
\end{array}$$

The same kind-indexed type, $\mathit{GMap}\langle\!\langle\kappa\rangle\!\rangle\ t\ t$, determines the types of instances of the generic functions. For our example type, $\mathit{Tree} :: \star \to \star$, we expect

$$\mathit{gmap}\langle \mathit{Tree}\rangle :: (a \to b) \to \mathit{Tree}\ a \to \mathit{Tree}\ b\,.$$

Notice that for a functor of one argument, such as *Tree*, the instance of *gmap* requires one function argument for transforming the values of type a into b. For a two argument functor, such as `:+:` above, two function arguments are required, one for each type parameter.

A generic function can be written using cases for arbitrary *named types*, i.e., for simple types or type constructors of any kind. Generic definitions therefore need not have cases for all the type constructors which form the structure types, nor need they be limited to just those. If a case for a different type is present in a generic definition, then this overrides the default behaviour. Apart from giving a distinct behaviour for a specific type, such lines can also be employed to provide a more efficient implementation for a particular type constructor. For instance, the *gmap* function could be made to use the (probably optimised) predefined *map* function on lists by adding the line

$$gmap\langle[]\rangle\ gmapA\ xs = map\ gmapA\ xs\,.$$

2.2 Generic types

In addition to generic functions, Generic H∀SKELL also supports types that are defined over the structure of datatypes. Similar in form to generic functions, a generic type definition consists of multiple cases for different named types, each resembling a Haskell **type** or **newtype** declaration. Generic types, also called *type-indexed types*, are described in depth in (Hinze et al., 2002), with several examples of their use, and are therefore not the focus of this paper.

2.3 Inside the Generic H∀SKELL compiler

Generic H∀SKELL compiles modules written in an enriched Haskell syntax. A Generic H∀SKELL module may contain, in addition to regular Haskell code, definitions of generic functions, kind-indexed types, and type-indexed types, as well as applications of these to types or kinds. The compiler translates a Generic H∀SKELL module into an ordinary Haskell module by performing a number of tasks:

- translating generic definitions into Haskell code;
- translating calls to generic functions into an appropriate Haskell expression; and
- specialising generic entities to the types at which they are applied (Consequently no type information is passed around at run-time).

In addition, the compiler generates structure types for all datatypes, together with functions that allow conversion between a datatype and its structure type.

Generic H∀SKELL programs may consist of multiple modules. Generic functions defined in one module may be imported into and reused in

other modules. Generic H∀SKELL comes with a library that provides a collection of common generic functions, among which are those usually generated by means of Haskell's **deriving** mechanism.

2.3.1 Translation of Generic Functions. In general, a generic function definition consists of a type signature referring to a kind-indexed type, and a collection of cases indexed by named types.

Each of the cases is translated into one ordinary Haskell function definition. The generated function is given a name which depends upon the generic function's name and the type name for which it is defined.[2] The special argument for a *Con* or *Lab* case is turned into an extra argument to the generated Haskell function, of type *ConDescr* or *LabDescr*, respectively. Both *ConDescr* and *LabDescr* are abstract types, for which Generic H∀SKELL provides a collection of functions to query information such as the name of the constructor or label. As an example, the `:+:` and the *Con* cases of the *gmap* function are translated to

$$
\begin{array}{l}
gmap__Sum :: (a \to c) \to (b \to d) \to a \texttt{ :+: } b \to c \texttt{ :+: } d \\
gmap__Sum\ gmapA\ gmapB\ (Inl\ a) \quad = \quad Inl\ (gmapA\ a) \\
gmap__Sum\ gmapA\ gmapB\ (Inr\ b) \quad = \quad Inr\ (gmapB\ b) \\
gmap__Con :: ConDescr \to (a \to b) \to Con\ a \to Con\ b \\
gmap__Con\ \ c\ gmapA \qquad (Con\ a) \quad = \quad Con\ (gmapA\ a)\,.
\end{array}
$$

The type signatures are generated internally by expanding the function's kind-indexed type.

2.3.2 Calls to Generic Functions. A call to a generic function *poly* in the source code takes the form

$$poly\langle T \rangle$$

where T is a type expression that can be either a named type or an application of one type expression to another.

One of the fundamental ideas of MPC-style generic definitions is to interpret type application as value application. In other words, for type expressions $F :: \kappa \to \nu$ and $A :: \kappa$, the equation

$$poly\langle F\ A \rangle = poly\langle F \rangle\ (poly\langle A \rangle)$$

holds. Knowing this fact, calls to generic functions can always be rewritten to an application expression containing several calls to the same

[2] The names assigned by the current implementation are additionally prefixed to prevent name clashes and differ from the names used in this paper.

generic function having only named types as arguments. For example, the call

$$gmap\langle Either\ [String]\rangle$$

is rewritten as

$$gmap\langle Either\rangle\ (gmap\langle[\,]\rangle\ (gmap\langle String\rangle))\,.$$

Calls that refer to named types are replaced by a call to the appropriate specialised function, such as $gmap__Either$.

2.3.3 Specialisation of Generic Functions. When a generic function is specialised to a named type, the compiler first checks whether the function in question has a case defined for that particular named type. If one is present, then it is used. The presence of a special case for a user-defined datatype thus overrides the standard behaviour of specialisation. In the absence of a special case, the compiler proceeds in two steps:

- the generic function is specialised to the named type's structure type; and
- the resulting specialised function is converted using a generic wrapper to a specialisation for the original named type.

Recall that a structure type is defined as a type synonym. Now a call to a generic function on a structure type can be viewed as a call on the right-hand side of that synonym. Hence, the specialisation of the generic function is again reduced to calls to named types as described in Section 2.3.2.

For example, specialising *gmap* to $Tree^{\circ}$ results in

$$\begin{array}{l} gmap__Tree^{\circ} \quad :: \quad (a \to b) \to Tree^{\circ}\ a \to Tree^{\circ}\ b \\ gmap__Tree^{\circ}\ a \quad = \\ \quad gmap__Con\ a \\ \quad\quad `gmap__\texttt{:+:}` \\ \quad gmap__Con\ (gmap__Lab\ a\ `gmap__\texttt{:*:}`\ gmap__Lab\ (gmap__Tree\ a) \\ \quad\quad\quad\quad\quad\quad `gmap__\texttt{:*:}`\ gmap__Lab\ (gmap__Tree\ a))\,. \end{array}$$

The type specialisation process is guaranteed to terminate, since structure types are only generated for the top-level of a type and the number of named types in a program is finite.

These specialisations work for a structure type rather than the original type. Fortunately, the isomorphisms mapping between a type and its

structure type are straightforward to generate. For our example, the function

$$bimap__Tree :: (Tree^{\circ}\ a \to Tree^{\circ}\ b) \to (Tree\ a \to Tree\ b)$$

is firstly generated as a lifted isomorphism, then $gmap__Tree$, which corresponds to $gmap\langle Tree\rangle$, becomes

$$\begin{array}{lcl} gmap__Tree & :: & (a \to b) \to Tree\ a \to Tree\ b \\ gmap__Tree\ a & = & bimap__Expr\ (gmap__Tree^{\circ}\ a)\,. \end{array}$$

These isomorphisms can be lifted to arbitrary types generically, and then used to wrap the function generated for structure types to convert it into a function for the original type. This technique is described in detail in (de Wit, 2002) and (Hinze, 2000a).

3. Contemporary Generic H∀SKELL

In the previous section we described the basic features of Generic H∀SKELL, which are to a high degree based on work published by Hinze (Hinze, 2000c; Hinze, 2000a). Recently a number of extensions for Generic H∀SKELL have been designed. These extensions, which have been implemented in the current version of our compiler, constitute the contribution of this paper and are described in this section.

The first, *default cases*, is a method to ease the programming of similar generic functions. A new generic function may be based on a previous one, inheriting the definitions for all cases that are not explicitly overridden.

Secondly, *constructor cases* allow generic function cases to be defined not only for named types, but also for particular constructors. This, at the first glance, may seem not particularly generic, but it happens to be useful in situations similar to those which require special cases to be defined for particular types, for example, when defining a traversal over a datatype with a large number of constructors which requires special action only at certain constructors.

Furthermore, we introduce *generic abstractions*, which allow generic functions to be defined by abstracting a type variable out of an expression which may involve generic functions.

All these extensions work well together, allowing some applications that are not among the typical generic programming examples. Along with the extensions, we provide demonstrations of their possible use, and a larger example showing their interaction.

3.1 Default cases

Many generic functions follow some commonly occurring pattern for most type constructors. Two such examples are *map-like* and *crush-like* functions (also called type-preserving and type-unifying traversals (Lämmel et al., 2000)) which, respectively, perform a computation during the traversal of a data structure while preserving the type, and collect information from the data structure into a value of a new type. Such functions exhibit certain general behaviour which differs for just a few types or constructors. Rather than duplicate the bulk of such definitions, Generic H∀SKELL introduces *default cases* so that the mechanics of the traversal can be written as one generic function, and then copied with modification or extension into other generic functions. Default cases are best described by example.

The following code forms the basis of a crush-like function which collects a list of values of type a from some datatype. As it currently is, this function can only return the empty list, but it provides the basic traversal which can be used in other functions.

$$
\begin{array}{lllll}
\textbf{type}\ \mathit{Collect}\langle\!\langle\star\rangle\!\rangle\ t & & & = & t \to [a] \\
\textbf{type}\ \mathit{Collect}\langle\!\langle\kappa \to \nu\rangle\!\rangle\ t & & & = & \forall u\,.\,\mathit{Collect}\langle\!\langle\kappa\rangle\!\rangle\ u \\
 & & & & \qquad \to \mathit{Collect}\langle\!\langle\nu\rangle\!\rangle\ (t\ u) \\
\mathit{collect}\langle t :: \kappa\rangle & & & :: & \mathit{Collect}\langle\!\langle\kappa\rangle\!\rangle\ t \\
\mathit{collect}\langle \mathit{Unit}\rangle & & \mathit{Unit} & = & [\,] \\
\mathit{collect}\langle \texttt{:+:}\rangle & \mathit{mA}\ \mathit{mB} & (\mathit{Inl}\ a) & = & \mathit{mA}\ a \\
\mathit{collect}\langle \texttt{:+:}\rangle & \mathit{mA}\ \mathit{mB} & (\mathit{Inr}\ b) & = & \mathit{mB}\ b \\
\mathit{collect}\langle \texttt{:*:}\rangle & \mathit{mA}\ \mathit{mB} & (a\ \texttt{:*:}\ b) & = & \mathit{mA}\ a +\!\!+ \mathit{mB}\ b \\
\mathit{collect}\langle \mathit{Con}\ c\rangle & m & (\mathit{Con}\ b) & = & m\ b \\
\mathit{collect}\langle \mathit{Char}\rangle & & c & = & [\,]
\end{array}
$$

Now suppose we are working with the following datatypes and wish to define a function which collects values of type *Var*, treating them as a set rather than simply concatenating them together into a list.

$$
\begin{array}{lcll}
\textbf{data}\ \mathit{Var} & = & \mathit{V} & \mathit{String} \\
\quad \textbf{deriving}\ \mathit{Eq} & & & \\
\textbf{data}\ \mathit{Type} & = & \mathit{TVar} & \mathit{Var} \\
 & | & \mathit{Arrow} & \mathit{Type}\ \mathit{Type} \\
\textbf{data}\ \mathit{Expr} & = & \mathit{Var} & \mathit{Var} \\
 & | & \mathit{App} & \mathit{Expr}\ \mathit{Expr} \\
 & | & \mathit{Lambda} & (\mathit{Var}, \mathit{Type})\ \mathit{Expr} \\
 & | & \mathit{Let} & (\mathit{Var}, \mathit{Type})\ \mathit{Expr}\ \mathit{Expr}
\end{array}
$$

We can do this by adapting the function *collect*. First we need a more specific type:

$$\begin{array}{lcl}
\textbf{type}\ \mathit{VarCollect}\langle\!\langle\star\rangle\!\rangle\ t & = & t \to [\,\mathit{Var}\,] \\
\textbf{type}\ \mathit{VarCollect}\langle\!\langle\kappa \to \nu\rangle\!\rangle\ t & = & \forall u\,.\ \mathit{VarCollect}\langle\!\langle\kappa\rangle\!\rangle\ u \\
 & & \quad \to \mathit{VarCollect}\langle\!\langle\nu\rangle\!\rangle\ (t\ u)\,.
\end{array}$$

We can now write the desired function in a few lines (the order of the cases is irrelevant):

$$\begin{array}{lcl}
\mathit{varcollect}\langle t :: \kappa\rangle & :: & \mathit{VarCollect}\langle\!\langle\kappa\rangle\!\rangle\ t \\
\mathit{varcollect}\langle \mathit{Var}\rangle\ v & = & [\,v\,] \\
\mathit{varcollect}\langle \texttt{:*:}\rangle\ mA\ mB\ (a\ \texttt{:*:}\ b) & = & mA\ a \cup mB\ b \\
\mathit{varcollect}\langle c\rangle & = & \mathit{collect}\langle c\rangle\,.
\end{array}$$

The line $\mathit{varcollect}\langle c\rangle = \mathit{collect}\langle c\rangle$ is the default case, which has the effect of copying the code from *collect* into the new generic function *varcollect*. The line for $\mathit{varcollect}\langle \mathit{Var}\rangle$ specifies the desired additional functionality for type *Var*. The line for $\mathit{varcollect}\langle \texttt{:*:}\rangle$ overrides the functionality for `:*:`, using union instead of concatenation when accumulating the results.

Compilation is simple. For the function *varcollect* defined above the compiler internally generates the following definition, and then compiles this code as usual:

$$\begin{array}{lcl}
\mathit{varcollect}\langle t :: \kappa\rangle & :: & \mathit{VarCollect}\langle\!\langle\kappa\rangle\!\rangle\ t \\
\mathit{varcollect}\langle \mathit{Var}\rangle\ v & = & [\,v\,] \\
\mathit{varcollect}\langle \texttt{:*:}\rangle\ mA\ mB\ (a\ \texttt{:*:}\ b) & = & mA\ a \cup mB\ b \\
\mathit{varcollect}\langle \mathit{Unit}\rangle & = & \mathit{collect}\langle \mathit{Unit}\rangle \\
\mathit{varcollect}\langle \texttt{:+:}\rangle & = & \mathit{collect}\langle \texttt{:+:}\rangle \\
\mathit{varcollect}\langle \mathit{Con}\ c\rangle & = & \mathit{collect}\langle \mathit{Con}\ c\rangle \\
\mathit{varcollect}\langle \mathit{Char}\rangle & = & \mathit{collect}\langle \mathit{Char}\rangle\,.
\end{array}$$

The compiler keeps track of which named types the original generic function is defined for, and updates and extends this collection depending on the cases specified in the new generic function.

We can also reuse cases from the existing definition in the new generic definition, even cases which are overridden. The following function collects just term variables from our expressions, using the function $\mathit{termvar} :: \mathit{Var} \to \mathit{Bool}$ to determine whether a variable comes from the syntactic category of term variables:

$$\begin{array}{lcl}
\mathit{termcollect}\langle t :: \kappa\rangle & :: & \mathit{VarCollect}\langle\!\langle\kappa\rangle\!\rangle\ t \\
\mathit{termcollect}\langle c\rangle & = & \mathit{varcollect}\langle c\rangle \\
\mathit{termcollect}\langle \mathit{Var}\rangle\ v & = & \textbf{if}\ \mathit{termvar}\ v\ \textbf{then}\ \mathit{varcollect}\langle \mathit{Var}\rangle\ v\ \textbf{else}\ [\,]\,.
\end{array}$$

It is very important that the new function defined by means of the default case only retains references to the original function in the right-hand side of the cases, and that the specialisation mechanism proceeds using solely the new function.

As an example, look at the definition of *varcollect* again. Because there is no new case for `:+:`, the compiler will infer

$$varcollect\langle \texttt{:+:} \rangle \quad = \quad collect\langle \texttt{:+:} \rangle \,.$$

It would appear that *varcollect*, once called for the `:+:` case, will work precisely as *collect*. But this is not true! The specialisation mechanism will replace $varcollect\langle A \texttt{ :+: } B \rangle$ as follows:

$$\begin{array}{rcl} varcollect\langle A \texttt{ :+: } B \rangle & = & varcollect\langle \texttt{:+:} \rangle \; varcollect\langle A \rangle \; varcollect\langle B \rangle \\ & = & collect\langle \texttt{:+:} \rangle \; varcollect\langle A \rangle \; varcollect\langle B \rangle \,. \end{array}$$

Thus the arguments to the `:+:` case will remain calls to *varcollect*.

As a consequence, *varcollect* will collect variables in a data structure deeply, i.e. in a large system of mutually recursive data types all occurrences of the type *Var* will be found.

In the next section we modify *varcollect* to collect only the free occurrences of variables.

3.2 Constructor cases

The problem of *dealing with large bananas* is to write map-like or crush-like functions for datatypes with a *large number of constructors*, such as those which model the syntax of a real programming language (Lämmel et al., 2000). The anticipated behaviour of such functions for most constructors of a datatype is standard and can be coded using existing generic programming techniques. For certain constructors, however, something different may be required, as when collecting *free* variables, for example, which requires the variable scoping to be considered for constructors such as *Let* and *Lambda* which denote binding constructs.

To do this using the machinery described thus far would require a case for the type we are interested in, namely, *Expr*. But then we would have to write the details of the traversal for each constructor of this type – both the interesting cases and the plumbing – gaining nothing, especially when there are a large number of constructors of which only a few require special treatment. Thus the machinery described so far cannot deal effectively with this problem.

To address this problem, Generic H∀SKELL now allows cases for specific constructors to be written. Using these *constructor cases* a generic function can have special cases to deal with the constructors requiring special treatment. Again we illustrate with examples.

Our first example uses a copy line to extend the function for collecting term variables with a constructor case for both the *Let* and *Lambda* constructors of the *Expr* datatype to take account of variable scoping:

$$
\begin{array}{lcl}
\mathit{freecollect}\langle t :: \kappa \rangle & :: & \mathit{VarCollect}\langle\!\langle \kappa \rangle\!\rangle\ t \\
\mathit{freecollect}\langle c \rangle & = & \mathit{termcollect}\langle c \rangle \\
\mathit{freecollect}\langle\ \mathbf{case}\ \mathit{Lambda}\ \rangle\ (\mathit{Lambda}\ (v,t)\ e) & & \\
\qquad = & \mathit{filter}\ (\neq v)\ (\mathit{freecollect}\langle \mathit{Expr} \rangle\ e) & \\
\mathit{freecollect}\langle\ \mathbf{case}\ \mathit{Let}\ \rangle\ (\mathit{Let}\ (v,t)\ e\ e') & & \\
\qquad = & \mathit{freecollect}\langle \mathit{Expr} \rangle\ e & \\
\qquad\qquad & +\!\!+\ \mathit{filter}\ (\neq v)\ (\mathit{freecollect}\langle \mathit{Expr} \rangle\ e')\,. &
\end{array}
$$

The case *freecollect*⟨ **case** *Lambda* ⟩, for example, is a constructor case. This case will only be applied when the value of type *Expr* which is actually encountered has the form *Lambda* (v, t) e. The case has been written to exploit this knowledge. The case for *Let* is similar. Interestingly, when a constructor case produces a value, it need not produce a value with the same constructor, but only of the correct type. This comes in handy in our second example.

Specific constructor cases of a generic function can be called using the syntax *freecollect*⟨ **case** *Lambda* ⟩. We expect that this will be most useful when reusing existing code while overriding a constructor case.

As another example we give the following mini-optimiser for lambda expressions, following in part (Lämmel et al., 2000). The function *gmap* has the basic form of a map-like function. This can be extended with a number of constructor cases to produce the following map-like function which performs some minor optimisations on lambda expressions:

$$
\begin{array}{l}
\mathit{optimise}\langle t :: \kappa \rangle \quad :: \quad \mathit{GMap}\langle\!\langle \kappa \rangle\!\rangle\ t\ t \\
\mathit{optimise}\langle c \rangle \quad = \quad \mathit{gmap}\langle c \rangle \\
\mathit{optimise}\langle\ \mathbf{case}\ \mathit{Let}\ \rangle\ (\mathit{Let}\ (v,t)\ e\ e') \quad = \\
\qquad \mathit{optimise}\langle \mathit{Expr} \rangle\ \$\ (\mathit{App}\ (\mathit{Lambda}\ (v,t)\ (\mathit{optimise}\langle \mathit{Expr} \rangle\ e'))\ e) \\
\mathit{optimise}\langle\ \mathbf{case}\ \mathit{App}\ \rangle\ (\mathit{App}\ e\ e') \quad = \\
\qquad \mathbf{let}\ \mathit{oe} = \mathit{optimise}\langle \mathit{Expr} \rangle\ e\ \mathbf{in} \\
\qquad \mathbf{let}\ \mathit{oe}' = \mathit{optimise}\langle \mathit{Expr} \rangle\ e'\ \mathbf{in} \\
\qquad\quad \mathbf{case}\ \mathit{oe}\ \mathbf{of}\ (\mathit{Lambda}\ (x,t)\ b) \rightarrow \\
\qquad\qquad\qquad\qquad \mathbf{if}\ \mathit{not}\ (\mathit{elem}\ x\ (\mathit{freecollect}\langle \mathit{Expr} \rangle\ b)) \\
\qquad\qquad\qquad\qquad \mathbf{then}\ b \\
\qquad\qquad\qquad\qquad \mathbf{else}\ \mathit{App}\ \mathit{oe}\ \mathit{oe}'
\end{array}
$$

This code performs a bottom-up traversal of an expression, converting let expressions into an application of a lambda abstraction, and optimising applications of a function whose argument does not occur in the

body. One point of interest is that the *Let* case produces a constructor different from *Let*.

Typing. The type of a constructor case is the same as the case for the type from which the constructor comes. For example, the type of $optimise\langle Expr\rangle$ is $GMap\langle\!\langle\star\rangle\!\rangle\ Expr\ Expr$, that is, $Expr \to Expr$. Thus the type of $optimise\langle$ **case** *Lambda* $\rangle$ must also be $Expr \to Expr$. As described above, there is the added restriction that any data which this function accepts must have the given constructor, otherwise pattern matching will fail at run-time. This cannot be enforced in the current system. A type system with constructor subtyping (Barthe and Frade, 1999) could help here.

Compilation. Constructor cases are implemented using a slight modification of the structure type encoding and specialisation process introduced in Section 2. We demonstrate the scheme on our example *Expr* datatype. For each constructor, a type synonym is introduced, identifying the type of the constructor case with the type it is constructing:

$$\begin{array}{lcl}
\textbf{type}\ ConCase_Var & = & Expr \\
\textbf{type}\ ConCase_App & = & Expr \\
\textbf{type}\ ConCase_Lambda & = & Expr \\
\textbf{type}\ ConCase_Let & = & Expr\,.
\end{array}$$

Whereas the old translation transformed the sum and the product structure of the datatype in one go, the modified translation proceeds in two steps. Firstly, only the sum structure is revealed, making use of the new type synonyms for the constructor cases:

$$\begin{array}{rcl}
\textbf{type}\ Expr^{\circ} & = & Con\ ConCase_Var \\
 & :+: & Con\ ConCase_App \\
 & :+: & Con\ ConCase_Lambda \\
 & :+: & Con\ ConCase_Let\,.
\end{array}$$

Secondly, the types for the constructor cases are themselves given structure types reflecting the product structure of the constituent fields:

$$\begin{array}{lcl}
\textbf{type}\ ConCase_Var^{\circ} & = & Var \\
\textbf{type}\ ConCase_App^{\circ} & = & Expr\ :*:\ Expr \\
\textbf{type}\ ConCase_Lambda^{\circ} & = & (Var, Type)\ :*:\ Expr \\
\textbf{type}\ ConCase_Let^{\circ} & = & (Var, Type)\ :*:\ Expr\ :*:\ Expr\,.
\end{array}$$

The mappings to convert between a type and its structure type must also be adapted – in fact, simplified – to reflect the modified shape of

the structure type. In addition, we now need a mapping for each of the constructors to convert between the constructor type synonym and its structure type.

Let us look at the first part more closely: for the *Expr* datatype, the mappings between *Expr* and $Expr^\circ$ could naïvely be implemented by the following functions *from* and *to*:

$$
\begin{array}{lcl}
from & :: & Expr \to Expr^\circ \\
from\ e@(Var\ _) & = & Inl\ (Con\ e) \\
from\ e@(App\ _\ _) & = & Inr\ (Inl\ (Con\ e)) \\
from\ e@(Lambda\ _\ _) & = & Inr\ (Inr\ (Inl\ (Con\ e))) \\
from\ e@(Let\ _\ _\ _) & = & Inr\ (Inr\ (Inr\ (Con\ e)))
\end{array}
$$

$$
\begin{array}{lcl}
to & :: & Expr^\circ \to Expr \\
to\ (Inl\ (Con\ e@(Var\ _))) & = & e \\
to\ (Inr\ (Inl\ (Con\ e@(App\ _\ _)))) & = & e \\
to\ (Inr\ (Inr\ (Inl\ (Con\ e@(Lambda\ _\ _))))) & = & e \\
to\ (Inr\ (Inr\ (Inr\ (Con\ e@(Let\ _\ _\ _))))) & = & e\,.
\end{array}
$$

The sum structure is created and removed on top of the type, the value remains completely intact. But when a generic function changes a constructor, as in *optimise* above, we get a run-time pattern matching error in the *to* function, because the sum structure encountered does not match the constructor that is present in the value.

In the current system, we cannot enforce that the argument and result values have the same outermost constructor. On the other hand, the constructor rewrite functionality proves to be useful in many places, as the *optimise* example shows. Therefore, we relax the implementation of *to* in order to ignore the sum structure that is present, remove it and recover the value stored inside.

$$
\begin{array}{lcl}
to\ (Inl\ (Con\ e)) & = & e \\
to\ (Inr\ (Inl\ (Con\ e))) & = & e \\
to\ (Inr\ (Inr\ (Inl\ (Con\ e)))) & = & e \\
to\ (Inr\ (Inr\ (Inr\ (Con\ e)))) & = & e
\end{array}
$$

With these preparations made, specialisation can now be handled exactly as described in Section 2.3.3, where constructors are treated as if they were additional types.

If a function is specialised for type *Expr*, then a specialisation for $Expr^\circ$ will be generated together with a wrapper function that lifts the isomorphism between *Expr* and $Expr^\circ$ to the specialisation's type. But, the specialisation $Expr^\circ$ will now refer to the specialisations of the same

function for all four constructor type synonyms. Thus, specialisations for the four constructors must also be generated.

A constructor case in the generic function definition now plays the role of a special case for an ordinary named type: a constructor case takes the place of the specialisation, thereby overriding the default behaviour. Otherwise, the specialisation is generated in the usual way for the structure type of the constructor, together with another wrapper function.

In the function definition of *optimise*, there are two constructor cases present, one for both *Lambda* and *Let*. Thus, when *optimise* is specialised to *Expr*, the functions

$$\begin{aligned}
&\mathit{optimise__Expr}^{\circ}, \mathit{optimise__Expr} && \text{(for the type } \mathit{Expr}\text{)} ,\\
&\mathit{optimise__ConCase_Var}^{\circ}, \mathit{optimise__ConCase_Var} && \text{(for the constructor } \mathit{Var}\text{)} ,\\
&\mathit{optimise__ConCase_App}^{\circ}, \mathit{optimise__ConCase_App} && \text{(for the constructor } \mathit{App}\text{)}
\end{aligned}$$

are generated in the usual manner, whereas *optimise__ConCase_Lambda* and *optimise__ConCase_Let* will be taken directly from the definition of the two constructor cases.

3.3 Generic Abstraction

In addition to copy lines, Generic H∀SKELL offers another possibility to define generic functions in terms of other generic functions, namely using *generic abstractions.*

In a full generic function definition, one is often forced to be more general than one intends to be. For instance, it is impossible to write a generic function that does not have function type when applied to a type of kind $\star \to \star$. This is because the specialisation mechanism interprets abstraction and application on the type level as abstraction and application on the value level, and thus forces generic function types to follow a specific pattern.

Generic abstraction lifts all restrictions that are normally imposed on the type of a generic function. With generic abstraction we can define a function which takes a single type argument which can be used generically within the body of the function. The abstracted type parameter is, however, restricted to types of a fixed kind. The resulting function can be then applied to any type having the appropriate kind. Generic abstractions can be used to write variations, simplifications and special cases of other generic functions.

The syntax of generic abstractions is similar to ordinary generic definitions, with two important differences:

- the type signature is restricted to a fixed kind, and thus no kind variable is introduced; and
- there is exactly one case, which has a type variable as its type argument, rather than being indexed by constant named types.

Suppose now we want a function generic in types $f :: \star \to \star$ which takes a data structure containing monadic actions and performs the monadic action stored at each point in the data structure. This function, which we call *thread*, is a generalisation of the Haskell prelude function *sequence*, which has type $Monad\ m \Rightarrow [m\ a] \to m\ [b]$, from lists to arbitrary type constructors. It is defined using generic abstraction as follows:

$$\begin{array}{lcl} thread\langle f :: \star \to \star \rangle & :: & (Functor\ m, Monad\ m) \Rightarrow f\ (m\ a) \to m\ (f\ a) \\ thread\langle f \rangle & = & mapMl\langle f \rangle\ id \end{array}$$

This declares a function called *thread* which has type parameter f of kind $\star \to \star$. The right-hand side of this definition uses this type parameter when applying the generic function *mapMl*, which is a monadic map function from the Generic H∀SKELL library. We reproduce *mapMl* here for completeness:

$$\begin{array}{lcl} \textbf{type}\ MapM\langle\!\langle \star \rangle\!\rangle\ t_1\ t_2\ m & = & t_1 \to m\ t_2 \\ \textbf{type}\ MapM\langle\!\langle \kappa \to \nu \rangle\!\rangle\ t_1\ t_2\ m & = & \\ \quad \forall u_1\ u_2\ .\ MapM\langle\!\langle \kappa \rangle\!\rangle\ u_1\ u_2\ m \to MapM\langle\!\langle \nu \rangle\!\rangle\ (t_1\ u_1)\ (t_2\ u_2)\ m & & \\ mapMl\langle t :: \kappa \rangle :: (Functor\ m, Monad\ m) \Rightarrow MapM\langle\!\langle \kappa \rangle\!\rangle\ t\ t\ m & & \\ mapMl\langle Unit \rangle\ \ Unit & = & return\ Unit \\ mapMl\langle \texttt{:+:} \rangle\ mA\ mB\ (Inl\ a) & = & \textbf{do}\ \{ a_2 \leftarrow mA\ a; return\ (Inl\ a_2) \} \\ mapMl\langle \texttt{:+:} \rangle\ mA\ mB\ (Inr\ b) & = & \textbf{do}\ \{ b_2 \leftarrow mB\ b; return\ (Inr\ b_2) \} \\ mapMl\langle \texttt{:*:} \rangle\ mA\ mB\ (a\ \texttt{:*:}\ b) & = & \textbf{do}\ \{ a_2 \leftarrow mA\ a; b_2 \leftarrow mB\ b; \\ & & \quad return\ (a_2\ \texttt{:*:}\ b_2) \} \\ mapMl\langle Con\ c \rangle\ mA\ \ (Con\ a) & = & \textbf{do}\ \{ a_2 \leftarrow mA\ a; \\ & & \quad return\ (Con\ a_2) \}\ . \end{array}$$

Generic functions can be used to – at least partially – remove the need for type classes. Generic abstractions then help redefining functions that previously required a class constraint. For instance, the *Eq* and *Ord* type classes can be replaced by generic equality and ordering functions that can be applied to all types. Similarly, *Functor* can be replaced by the *gmap* function. Now, there are quite a number of useful functions that are usually implemented as ad-hoc polymorphic functions referring to

one of these classes. An example from the Haskell prelude is

$$lookup \quad :: \quad (Eq\ a) \Rightarrow a \to [(a, b)] \to Maybe\ b$$

Written using generic abstraction, this function can be implemented to make use of the generic equality function *equal*:

$$\begin{array}{lcl} glookup\langle a :: \star \rangle & :: & a \to [(a, b)] \to Maybe\ b \\ glookup\langle a \rangle\ key\ [\,] & = & Nothing \\ glookup\langle a \rangle\ key\ ((x, y) : xys) & & \\ \quad \mid equal\langle a \rangle\ key\ x & = & Just\ y \\ \quad \mid otherwise & = & glookup\langle a \rangle\ key\ xys\ . \end{array}$$

Compilation. We cannot naïvely compile a generic function defined via generic abstraction using the techniques described in Section 2.3.3. A specialisation to a type expression cannot always be reduced to a set of specialisations on named types only. In particular, assume that $F :: \star \to \star \to \star$ and $A :: \star$. We cannot apply the simplification rule

$$thread\langle F\ A \rangle = thread\langle F \rangle\ thread\langle A \rangle\ ,$$

since we have cases for neither $\star \to \star \to \star$ nor $\star$ in the definition of *thread*. Instead we produce a new function to denote $thread\langle F\ A \rangle$, and then, unravelling the definition

$$thread\langle F\ A \rangle = mapMl\langle F\ A \rangle\ id\ ,$$

we set the new function's right-hand side to be $mapMl\langle F\ A \rangle\ id$.

Recall that right-hand sides of generic abstractions may contain calls to other generic abstractions, not only to regular generic functions. Furthermore, the type arguments of the calls on the right-hand side are not necessarily simpler than the type-argument on the left-hand side. Thus the translation mechanism for generic abstractions is not guaranteed to terminate. This does not seem a big problem in practice, though, because we do not yet know of any set of useful generic functions that call themselves recursively with type arguments of increasing complexity. In the current implementation, the number of iterations performed to determine the specialisations needed can be bounded. It remains to be investigated whether it would be more practical to impose restrictions on the right-hand sides of generic abstractions so that termination could be guaranteed.

3.4 A larger example

As the final part of this section, we present a larger example consisting of multiple generic functions that interact and make use of all the features presented in the preceeding subsections.

Consider the problem of comparing two expressions up to alpha-equivalence: two expressions are equal if they can be transformed into each other only by renaming variables bound in let or lambda expressions.

One way of solving this problem is to introduce a normalisation step and then use standard equality on the normalised values. We choose to transform expressions to use de Bruijn indices for their bound variables: instead of a name, each bound variable is replaced by a number that indicates how many binding constructs are between the binding occurrence and the variable.

Recall the datatypes used before: variables occurring in expressions are of type

$$\textbf{data}\ \mathit{Var} = \mathit{V}\ \mathit{String}\ \textbf{deriving}\ \mathit{Eq}\,.$$

We could now extend this type to allow both free variables (strings) and bound variables (numbers) to be represented. However, for simplicity we will keep the original type and simply replace bound variables with string representations of numbers. In the binding occurrences the variables are superfluous after the numbering, hence there is no longer any need to store the name. We will model this by storing the empty string. As we plan to do no more with the numbered expression than compare them, this simplified representation satisfies our requirements.

The function computing the de Bruijn numbering from an expression is derived from the monadic map and uses a simple state monad to keep track of the list of currently known bound variables. The position of a variable in the list indicates its nesting level.

We assume the following interface to the abstract state monad type:

$$\begin{array}{l}
\textbf{data}\ \mathit{State}\ s\ a \qquad \text{-- abstract}\\
\textbf{instance}\ \mathit{Monad}\ (\mathit{State}\ s)\\
\textbf{instance}\ \mathit{Functor}\ (\mathit{State}\ s)\\
\mathit{readstate} \;::\; \mathit{State}\ s\ s\\
\mathit{writestate} \;::\; s \to \mathit{State}\ s\ ()\\
\mathit{runstate} \;::\; \mathit{State}\ s\ a \to s \to a\ .
\end{array}$$

We now implement *deBruijn* by extending *mapMl* using a default case. We implement only three additional cases. For a variable, check whether it is currently bound. If it is, replace by the correct index. For lambda and let expressions, the newly bound variable is added to the state before

proceeding to the body of the construct, and removed again afterwards.

```
deBruijn⟨t :: κ⟩          ::  MapM⟪κ⟫ t t (State [String])
deBruijn⟨t⟩               =   mapMl⟨t⟩
deBruijn⟨Var⟩ (V a)       =   do s ← readstate
                                 case findIndex (== a) s of
                                   Nothing → return $ V a
                                   Just n → return $ V $ show (n + 1)
deBruijn⟨ case Lambda ⟩ (Lambda (V a, t) e)
                          =   do e' ← boundin⟨Expr⟩ a e
                                 return (Lambda (V "", t) e')
deBruijn⟨ case Let ⟩ (Let (V a, t) e1 e2)
                          =   do e1' ← deBruijn⟨Expr⟩
                                 e2' ← boundin⟨Expr⟩ a e2
                                 return (Let (V "", t) e1' e2') .
```

The helper function *boundin* is implemented using generic abstraction, although we only use it with type *Expr* here. It calls *deBruijn*, first temporarily adding a variable name to the state.

```
boundin⟨a :: ⋆⟩           ::  String → a → State [String] a
boundin⟨a⟩ var body       =   do s ← readstate
                                 writestate (a : s)
                                 body' ← deBruijn⟨a⟩ body
                                 writestate s
                                 return body' .
```

Note that even though we explicitly refer to *Var* as well as the constructors *Lambda* and *Let* of *Expr*, this is still a generic function. Programming this way has its advantages: not only is the code less sensitive to future extensions or modifications in the expression datatype, we also get the same function for free working on types containing values of type *Expr*, from simple lists or trees of expressions to "real" extensions such as statements or function declarations.

Once again making use of generic abstraction, we hide the state monad by supplying an initial empty list of bound variables.

```
norm⟨a :: ⋆⟩    ::  a → a
norm⟨a⟩ x       =   runstate (deBruijn⟨a⟩ x) [] .
```

Finally, we implement the equality up to alpha-conversion using normal generic equality:

```
alphaequal⟨a :: ⋆⟩      ::  a → a → Bool
alphaequal⟨a⟩ x1 x2     =   equal⟨a⟩ (norm⟨a⟩ x1) (norm⟨a⟩ x2) .
```

Alternatively, we could implement *alphaequal* using a default case, extending the original equality function, using the relaxed notion of equality only for occurrences of the *Expr* datatype:

$$\begin{array}{lcl} alphaequal\langle t :: \kappa \rangle & :: & Equal\langle \star \rangle\ t\ t \\ alphaequal\langle t \rangle & = & equal\langle t \rangle \\ alphaequal\langle Expr \rangle & = & equal\langle Expr \rangle \\ & & \quad (norm\langle Expr \rangle\ x_1)\ (norm\langle Expr \rangle\ x_2)\ . \end{array}$$

4. Limitations and Future Work

Although the enhancements described in this paper seem to make Generic H∀SKELL suitable for practical programming, room for improvement still remains. In this section we pinpoint a few problems that will be subject of further investigation and research.

(Mutually) recursive generic functions and types. Sometimes a generic function makes use of one or more other generic functions. We have seen examples in the form of generic abstractions, but what about a full-fledged generic function with multiple cases which refers to another generic using a variable type argument? Consider a generic function that collects information from a value of a generic type, which descends further into the branch of a product case only when a certain condition holds:

$$\begin{array}{l} testcollect\langle :*: \rangle\ tcA\ tcB\ (a :*: b) = \\ \quad \textbf{let}\ rA = \textbf{if} \ldots \textbf{then}\ tcA\ a\ \textbf{else}\ [\,] \\ \quad\quad\ rB = \textbf{if} \ldots \textbf{then}\ tcB\ b\ \textbf{else}\ [\,] \\ \quad \textbf{in}\ rA +\!\!+ rB\ . \end{array}$$

We want the test, indicated by the ellipses, to depend generically on the types of a and of b. But since we cannot access these types on the right-hand side, we cannot include a call to the generic function that performs the test.

This limitation stems from the fact that we are using MPC-style, where generic function definitions are seen as catamorphisms: the recursive calls of the function to the children are given as parameters, but neither the types of the children itself nor the calls to other functions are available on the right-hand side.

There is a solution: we can tuple different functions into a single bundle and define them together. Instead of getting the recursive calls for just one function, we then get a tuple containing all the required functions. Although a tenable solution, this approach is difficult to use by hand: the programs written in this style are confusing and error-

prone. For example, applying tupling to our fragment results in:

$$
\begin{array}{l}
\mathit{testcollect}\langle \mathtt{:*:} \rangle\ (tA, cA)\ (tB, cB)\ (a\ \mathtt{:*:}\ b) = \\
\quad (\mathit{test}\langle \mathtt{:*:} \rangle\ tA\ tB \\
\quad , \mathbf{let}\ rA = \mathbf{if}\ tA\ a\ \mathbf{then}\ cA\ a\ \mathbf{else}\ [\,] \\
\qquad\quad rB = \mathbf{if}\ tB\ a\ \mathbf{then}\ cB\ b\ \mathbf{else}\ [\,] \\
\quad\ \mathbf{in}\ rA +\!\!+ rB \\
\quad)\,.
\end{array}
$$

Another style of generic function definitions, also introduced by Hinze (Hinze, 2000b), removes the problem by making the type arguments of the named types in the function cases explicit. Thus, there is the possibility to syntactically refer to other functions. However, these so-called POPL-style function definitions bear a number of limitations that MPC definitions do not share, most notably the restriction to types of a single kind and the requirement that a different set of structure type constructors is used for each different kind.

We are working on a combination that allows a programming syntax similar to POPL-style, but uses MPC-style functions behind the scenes, thereby allowing the interaction of multiple generic functions without losing the generality offered by the current implementation.

Type inference. The usage of generic functions is less convenient than one could expect because it always requires the type arguments to be explicitly specified. This restriction could be lifted using some form of type inference. Another solution is to integrate generic functions with type classes as done in the generic extensions to the Glasgow Haskell Compiler (Hinze and Peyton Jones, 2001) or to Clean (Alimarine and Plasmeijer, 2001).

As long as we limit generic functions to specific kinds, the problem of assigning types is closely related to the type inference and checking of type classes and is therefore quite well understood. However, for MPC-style generic functions in their full generality, type inference becomes more complicated, involving functions applied at multiple kinds, having types of different shapes, and thus requires further research.

Grouping. In many situations, it would be desirable to quantify over a group of types. In the *gmap* function, for example, it would be more concise to be able to write that $\mathit{gmap}\langle T \rangle = \mathit{id}$, for *all* types $T :: \star$. Instead, though, one has to write a separate line for each base type, and when a new abstract type is added, the function definition must be extended.

One possible approach to solving this problem might be to use the Haskell type class system. One could allow cases of generic functions to be specified for all members of a class. The current implementation does not keep track of classes and instances though.

5. Related Work

The field of generic programming has grown considerably in recent years. Much of this work stems from so-called theories of datatypes, for example (Bird et al., 1996), which are often based on category theoretic notions such as initial algebras. These approaches focus on generic control constructs, such as folds, which are derived from datatypes. A good overview is available (Backhouse et al., 1999).

Other approaches more or less define functions inductively on the structure of types. The work of Hinze, upon which Generic H∀SKELL is based, is just one variant. Chen and Appel, for instance, describe an implementation of polytypic polymorphism using dictionary passing (Chen and Appel, 2001). They do not, however, give any indication of how to code polytypic functions, working instead with built-in polytypic functions such as pretty printing and equality test. Weirich and others (Weirich, 2002) (and the earlier work on intensional polymorphism (Crary et al., 1998), for example) employ a **typecase** construct which performs run-time tests on types to implement polytypic functions of a expressiveness similar to Hinze's. Ruehr's structural polymorphism (Ruehr, 1992) adopts similar type tests. Generic H∀SKELL distinguishes itself from these approaches as it does not require type interpretation at run-time. In Functorial ML the algorithm for *map*, for example, is defined using combinators to find the *data* upon which argument function will apply (Jay et al., 1998). While supporting type inference, Functorial ML programming is at a rather low level and lacks the simplicity of Hinze's approach.

The basic theoretical work has manifest itself in several programming languages and language extensions. Functorial ML and other work on shape theory has resulted in the programming language FISh (Jay, 1999). Based on initial algebra semantics, Charity (Cockett and Fukushima, 1992) automatically generates folds and so forth for datatypes on demand, but it otherwise does not allow the programmer to write her own generic functions. PolyP (Jansson and Jeuring, 1997) extends Haskell with a special construct for defining generic functions, also based on initial algebras. In addition to the cases we consider, the programmer must define cases for type composition and type recursion. Although PolyP permits type inference, it supports only regular data types, not

mutually recursive, multiparametered, nested types, or types containing function spaces.

The programming language Haskell supports **deriving** clauses for a certain number of built-in classes (*Eq*, *Show*, etc) (Peyton Jones et al., 1999). This facilitates the automatic overloading of certain functions for datatypes which have the deriving clause specified. Hinze and Peyton Jones explored an extension, following Hinze's earlier ideas (Hinze, 2000b), which integrated generics with Haskell's type class system (Hinze and Peyton Jones, 2001). This system suffers from some limitations due to the interaction with the type class system. G'Caml (Furuse, 2001a; Furuse, 2001b) presents a generic programming extension for O'Caml (Leroy et al., 2001). The proposal does not aim to cover all datatypes, and as such can be seen as a way of achieving Haskell-style overloading in O'Caml. The generic extension for Clean is also based on Hinze's work (Alimarine and Plasmeijer, 2001). This proposal is more closely integrated with the type class system, but does not include any of the extensions described here. Generic H∀SKELL, on the other hand, takes the approach of exploring generic programming in isolation, as an extension to the Haskell language.

In the earlier days of generic programming the number of example programs was rather small: equality tests, pretty printing and parsing, map, fold, and so forth. As research continues, more examples are being unearthed. Pretty printing and parsing were reincarnated as data compression (Jansson and Jeuring, 2001). Type-indexed datatypes allow many new possibilities: digital searching, pattern matching, and a generalised zipper (Hinze et al., 2002). Unification (Jansson and Jeuring, 1998), term rewriting (Jansson and Jeuring, 2000), and other program transformations can also be seen as generic in the term datatype.

Generic program transformation has been the realm over which Stratego reigns (Visser et al., 1999). But Stratego is untyped. A number of approaches have been proposed for doing program transformations in a typed environment, these include RhoStratego (Dolstra and Visser, 2001), updateable algebras (Lämmel et al., 2000), and typed generic traversals (Lämmel and Visser, 2002; Lämmel, 2002). We hope to achieve such expressiveness within our framework and have described in this paper some steps in this direction.

6. Conclusion

Generic H∀SKELL implements a new approach to generic programming due to Hinze (Hinze, 2000c) which promises to relieve Haskell programmers of the burden of writing the same functions over and over

again for different datatypes. Its main advantages are that generic definitions are simple to write and can be applied to Haskell 98 types of all kinds.

To gain better leverage from this technology, we have introduced a few new features to Generic H∀SKELL which are useful when writing generic functions. These extensions are copy lines, constructor cases, and generic abstraction. Although these are relatively small modifications to the existing machinery, their interaction seems to open up new application areas for Generic H∀SKELL.

As we experiment with Generic H∀SKELL and approach problems in different ways, we continually discover new ways of using the existing machinery and stumble upon new challenges. These we hope to address from both theoretical and practical perspectives in the near future.

The extensions described herein have been implemented in the Generic H∀SKELL compiler, which is freely available for Unix, Windows, and MacOS X based platforms from `http://www.generic-haskell.org`.

References

Alimarine, A. and Plasmeijer, R. (2001). A generic programming extension for Clean. In *Proceedings of the 13th International workshop on the Implementation of Functional Languages, IFL'01*, pages 257–278.

Backhouse, R., Jansson, P., Jeuring, J., and Meertens, L. (1999). Generic programming: An introduction. In Swierstra, S. D., Henriques, P. R., and Oliveira, J. N., editors, *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag.

Barthe, G. and Frade, M. J. (1999). Constructor subtyping. In Swierstra, D., editor, *ESOP'99*, volume 1576 of *LNCS*, pages 109–127. Springer-Verlag.

Bird, R., de Moor, O., and Hoogendijk, P. (1996). Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28.

Chen, J. and Appel, A. W. (2001). Dictionary passing for polytypic polymorphism. Technical Report TR-635-01, Princeton University.

Clarke, D., Hinze, R., Jeuring, J., Löh, A., and de Wit, J. (2001). The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Institute of Information and Computing Sciences, Utrecht University.

Cockett, R. and Fukushima, T. (1992). About Charity. Yellow Series Report No. 92/480/18, Dep. of Computer Science, Univ. of Calgary.

Crary, K., Weirich, S., and Morrisett, J. G. (1998). Intensional polymorphism in type-erasure semantics. In *Proceedings ICFP 1998: International Conference on Functional Programming*, pages 301–312. ACM Press.

de Wit, J. (2002). A technical overview of Generic Haskell. Master's thesis, Department of Information and Computing Sciences, Utrecht University.

Dolstra, E. and Visser, E. (2001). First-class rules and generic traversal. Technical Report UU-CS-2001-38, Institute of Information and Computing Sciences, Utrecht University.

Furuse, J. (2001a). G'Caml – O'Caml with extensional polymorphism extension. Project home page at `http://pauillac.inria.fr/~furuse/generics/`.

Furuse, J. (2001b). Generic polymorphism in ML. In *Journées Francophones des Langages Applicatifs*.

Hinze, R. (2000a). *Generic Programs and Proofs*. Habilitationsschrift, Bonn University.

Hinze, R. (2000b). A new approach to generic functional programming. In *Conference Record of POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132. ACM Press.

Hinze, R. (2000c). Polytypic values possess polykinded types. In Backhouse, R. and Oliveira, J. N., editors, *Mathematics of Program Construction*, volume 1837 of *LNCS*, pages 2–27. Springer-Verlag.

Hinze, R., Jeuring, J., and Löh, A. (2002). Type-indexed data types. In *Proceedings of the 6th Conference on Mathematics of Program Construction (MPC 2002)*. Springer Verlag. To appear.

Hinze, R. and Peyton Jones, S. (2001). Derivable type classes. In Hutton, G., editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science. The preliminary proceedings appeared as a University of Nottingham technical report.

Jansson, P. and Jeuring, J. (1997). PolyP — a polytypic programming language extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press.

Jansson, P. and Jeuring, J. (1998). Functional pearl: Polytypic unification. *Journal of Functional Programming*, 8(5):527–536.

Jansson, P. and Jeuring, J. (2000). A framework for polytypic programming on terms, with an application to rewriting. In Jeuring, J., editor, *Workshop on Generic Programming 2000, Ponte de Lima, Portugal, July 2000*, pages 33–45. Utrecht Technical Report UU-CS-2000-19.

Jansson, P. and Jeuring, J. (2001). Polytypic data conversion programs. *Science of Computer Programming*. In press.

Jay, C., Bellè, G., and Moggi, E. (1998). Functorial ML. *Journal of Functional Programming*, 8(6):573–619.

Jay, C. B. (1999). Programming in FISh. *International Journal on Software Tools for Technology Transfer*, 2:307–315.

Lämmel, R. (2002). Typed Generic Traversal With Term Rewriting Strategies. Technical report. 85 pages; To appear in the Journal of Logic and Algebraic Programming; First published as CWI Technical Report SEN-R0122 in August 2001.

Lämmel, R. and Visser, J. (2002). Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*. Springer-Verlag.

Lämmel, R., Visser, J., and Kort, J. (2000). Dealing with Large Bananas. In Jeuring, J., editor, *Proceedings of WGP'2000, Technical Report, Universiteit Utrecht*, pages 46–59.

Leroy, X. et al. (2001). *The Objective Caml system release 3.02, Documentation and user's manual*. Available from `http://caml.inria.fr/ocaml/htmlman/`.

Peyton Jones, S., Hughes, J., et al. (1999). Haskell 98 — A non-strict, purely functional language. Available from `http://haskell.org`.

Ruehr, F. (1992). *Analytical and Structural Polymorphism Expressed Using Patterns Over Types*. PhD thesis, University of Michigan.

Visser, E., Benaissa, Z.-e.-A., and Tolmach, A. (1999). Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26. Proceedings of the International Conference on Functional Programming (ICFP'98).

Weirich, S. (2002). Higher-order intensional type analysis. In Métayer, D. L., editor, *Programming Languages and Systems: 11th European Symposium on Programming, ESOP 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002*. To appear.